Chapter 1

Seamless: A Reflective Middleware for Pharo (DRAFT)

by Nick Papoulias

Seamless [1] is a reflective communication middleware for Pharo that aims to facilitate the prototyping of distributed applications. It provides developer friendly abstractions and syntactic sugar for an *out-of-the-box* Pharo to Pharo communication experience. After prototyping your application, Seamless lets you easily profile and tweak communication settings such as distribution and serialization policies for optimizing performance and avoid common distribution caveats. This Chapter covers version 0.4 of Seamless for Pharo 3.0 and will be updated frequently towards version 1.0 (stable release).

1.1 Wait: Reflective What?

The first thing we do, let's kill all the lawyers – from "Henry VI", Shakespeare

A communication middleware is a networking solution (such as a library or a framework) that aims to hide some of the complicated networking setup (low-level details of socket management, remote method invocation, naming etc) from the developer. A reflective communication middleware is the more dynamic variant of such a system whose implementation relies on run-

[1] http://ss3.gemstone.com/ss/Seamless.html

time reflection and can thus be itself adapted and configured at runtime [2] . Seamless is such a reflective middleware for Pharo and as a project can be considered a descendant of Remote Smalltalk [3] which itself was preceded by projects such as OpenCorba [4] and Distributed Smalltalk [5]. It was born as an engineering prerequisite for a research prototype on remote debugging and has since be used on and off internally by other RMoD projects at INRIA.

From the point of view of the Pharo developer Seamless aspires to be - eventually - what RMI and DCOM is for Java and the .NET platforms respectively, while leveraging the reflective and dynamic nature of our environment. Seamless may be distinguished from other solutions in that:

- It targets the Pharo environment and strives to integrate with its core facilities (ie available serialization solutions like Fuel, proxy implementations like Ghost etc)
- It provides abstractions, syntactic carbohydrates (sugar) and programming facades to make the prototyping of distributed applications feel *out-of-the-box*. In a nutshell Seamless is biased towards Rapid Application Prototyping.
- Under the hood it tries to reify every single part of the object distribution process and its policies, so as to allow extensibility as well as easy profiling and fine-tuning.

1.2 Enough Said: Death to Sockets!

Installation

Seamless has been ported to Pharo 3.0. To install Pharo on your system follow the online instructions here or here. To install Seamless and start playing around evaluate the following code-snippet in your Workspace:

```
Gofer it
   url: 'http://ss3.gemstone.com/ss/Seamless';
   package: 'ConfigurationOfSeamless';
   load.

((Smalltalk at: #ConfigurationOfSeamless) project version: '0.4') load.
```

[2] For more info on reflective middleware, you can read this overview by Fabio Kon et al: http://www.inf.ufg.br/~fmc/papers/CACM-ReflectiveMiddleware-no-access.pdf

[3] http://www.squeaksource.com/rST.html

[4] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.79.1783

[5] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.7323

Alternatively Seamless can be also loaded directly from the command line as follows:

```
./pharo-ui Pharo.image config \
        http://ss3.gemstone.com/ss/Seamless \
        ConfigurationOfSeamless \
        --install=0.4
```

Then in order to be able to initiate and respond to requests from other images evaluate:

```
aDeamon := SeamlessDeamon
      newDefaultWithGlobalAccess
      startOn: 8080.
```

which again can be done directly from the command-line at start-up. To sum-up here is how you can download pharo, load Seamless and start the daemon in your preferred port in one go:

```
wget -O- get.pharo.org | bash;
./pharo-ui Pharo.image config \
        http://ss3.gemstone.com/ss/Seamless \
        ConfigurationOfSeamless \
        --install=0.4;
./pharo-ui Pharo.image eval "SeamlessDeamon newDefaultWithGlobalAccess
      startOn: 8080"
```

In order to follow the example in this chapter you will be needing two images, listening to port 8080 (**peer1** from now on) and 8081 (**peer2** from now on) respectively.

Hallo Transcript!

Once you have your peers set-up you can start sending messages from one image to another. This section will step-by-step dissect these remote msg-sends by showing you:

- How you get a reference to a remote receiver.
- What happens when you pass different kind of objects as arguments.
- How values from these message-sends are returned.

along the way we will also see how to use the SeamlessLogger, which doubles as a profiler for remote messaging.

Fetching Remote References

The simplest example possible would be to print the customary 'hallo world' or a sequence of numbers from peer1 to the transcript of peer2.

There are a lot of different ways to do that with Seamless let's see some of them (from peer1):

```
"sugar 1"
remoteTranscript := (Transcript from: '127.0.0.1:8081').
remoteTranscript open.
remoteTranscript show: 'Hallo World !'.
1 to: 100 do: [:i |
   remoteTranscript show: i; cr.
].
```

As you can see the only difference with printing locally is on line 2 when we retrieve the remote transcript instead of using the local Transcript of peer1. We will come back to the trade-offs introduced to your code by this *transparency* between local and remote execution. In the meantime here is some other ways you can retrieve a remote object and start sending messages to it with Seamless:

```
"sugar 2"
remoteTranscript := '127.0.0.1:8081' globalAt: #Transcript.
"sugar 3"
remoteEnvironment := '127.0.0.1:8081' asRemoteEnvironment.
remoteTranscript := remoteEnvironment at: #Transcript.
"sugar 4 (alt 1)"
remoteTranscript := (Smalltalk from: '127.0.0.1:8081') globals at: #Transcript.
```

This is of course all syntactic sugar with the seamless daemon and protocol working behind the scene to manage your remote references. You can even use these as a base to introduce your own abstractions for fetching remote objects.

Passing Arguments By Value

Let's now re-run our Transcript example (shown above), only this time we will use the SeamlessLogger (on both peers) to take a pick behind the scenes:

```
SeamlessLogger new.
```

Figure 1.1 shows the loggers on peer1 (left) and peer2 (right) after printing "1 to: 100" on the Transcript of peer2 as we did earlier. On the upper left side of each logger we can see the list of *daemons* in each peer. Multiple daemons with different configurations (communication and serialization protocols and policies) can co-exist in each peer. While each one of

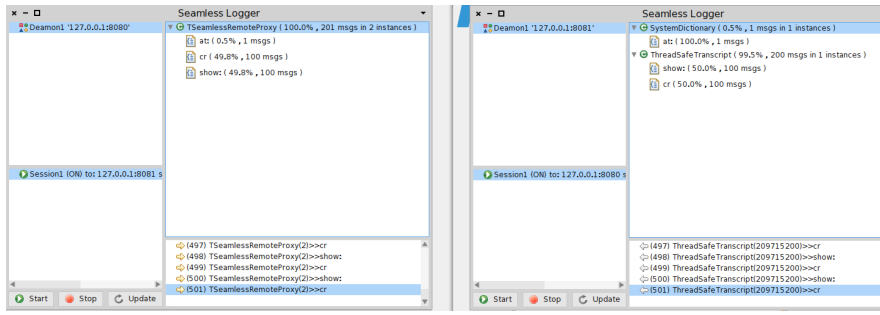

Figure 1.1: Remote Printing of Basic Objects (LOG).

these daemons can connect to multiple remote peers initiating a one-to-one *session* with each one (lower left side of each logger). On each of these *sessions* we can start/stop logging (stopping also clears the log) and update the log-views (upper right and lower right side of each logger) even while the communication is taking place. Automatic updating of the log is currently disabled in order to minimize the slow-down introduced by the logging facilities.

The log-views for our example inform us that:

- **Peer1 (Upper-Log)**: 2 instances of TSeamlessRemoteProxy (ie remote objects) received all *local* messages (100% of them) that amounted to 201 messages in total. 0.5% (ie 1 out of the 201) of those messages send to TSeamlessRemoteProxy instances where #at:, 49.8% (ie 100 messages) where #cr, while the rest 49.8% (ie 100 messages) were #show.
- **Peer1 (Lower-Log)**: All messages were outgoing (outgoing arrow), with each line also displaying:
  - the message-id counting from the start of the session
  - the class, with a unique qualifier for each instance that received the message under parenthesis (for proxies this is the proxy-id for local objects this is their hash)
  - the selector of the message
- **Peer2 (Upper-Log)**: As expected it was an instance (local to peer2) of the SystemDictionary class that received the #at: (to retrieve the remoteTranscript) while all other messages (200 of them) were send to 1 instance of the ThreadSafeTranscript, and were divided among #show and #cr sends.
- **Peer2 (Lower-Log)**: All messages were incoming (ingoing arrow), with each line displaying as before the message-id, the class with a

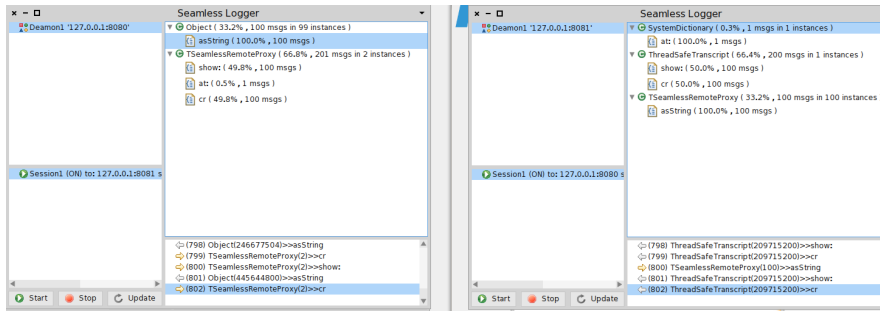

Figure 1.2: Remote Printing of Composite Objects (LOG).

unique qualifier for each instance and the selector.

Passing Arguments By Reference

So far so good, no surprises here as far as remote communication goes this is the simplest example possible. Let's try something else now:

```
1 remoteTranscript := '127.0.0.1:8081' globalAt: #Transcript.
2 aLocalObject := Object new.
3 (1 to: 100) do: [:i |
4   remoteTranscript show: aLocalObject; cr.
5 ].
```

The only thing we have changed here (on line 4) is the argument passed to the message #show: that the remoteTranscript receives. The code behaves as expected (ie aLocalObject is printed on the remoteTranscript as previously), but this time *behind the scenes* something different is going on. As shown in Figure 1.2 our **explicit** call to the remoteTranscript generated **implicit** proxies and incomming traffic from the other side. More specifically aLocalObject was implicitely proxied to peer2 because it was passed as an argument to #show, and then this new proxy generated traffic because it received the message #asString from the Transcript on peer2.

Note also here that because of this implicit *proxying*, although the code seems to send 200 messages (100 shows and 100 cr) the traffic that was generated in both directions in total was 300 messages. So why this difference, what changed ? The answer is that the argument to #show: changed from a Number to a composite Object. More specifically it changed from a terminal instance that **can be copied** when passed as an argument, to an instance that is not terminal but *potentially* points to other objects. This latter instance **should be referenced** (ie proxied) otherwise (if copied) it will not remain in sync with the other side. **Should** is the operative word here, cause at the end

of the day it is you (the programmer) that decides what gets *proxied* (and remains in sync) and what gets *copied* as terminal information (be it terminal instances such as numbers or even composite objects with their whole object graph).

By default Seamless chooses for you a handful of classes (including all their subclasses) to be passed or returned by value (ie copied) while all other local arguments to a remote call (or return values) will be passed by reference (implicitly creating proxies on the other side). This is done because while prototyping, you shouldn't - ideally - care. Knowing what Seamless does for you though will help you code more efficiently. As we said a simple rule that you can guide you is that by default *basic objects* such as numbers and strings are passed by value while more *composite objects* are passed by reference. The global default daemon of Seamless passes the instances of the following classes (and of their subclasses) by value:

- Boolean
- Character
- Number
- String
- Symbol
- UndefinedObject
- Color
- Point

Changing Policies on The Fly

Of course the above list can be changed dynamically either while setting up your daemon or even (if needed) while your application is running. Even more conveniently you can programmatically choose to serialize a previously proxied object or *force* the serialization of a local object to the other side on a per instance and per message basis. For our example this would be done as follows:

```
1 remoteTranscript := '127.0.0.1:8081' globalAt: #Transcript.
2 aLocalObject := Object new.
3 (1 to: 100) do: [:i |
4   remoteTranscript show: aLocalObject byValue; cr.
5 ].
```

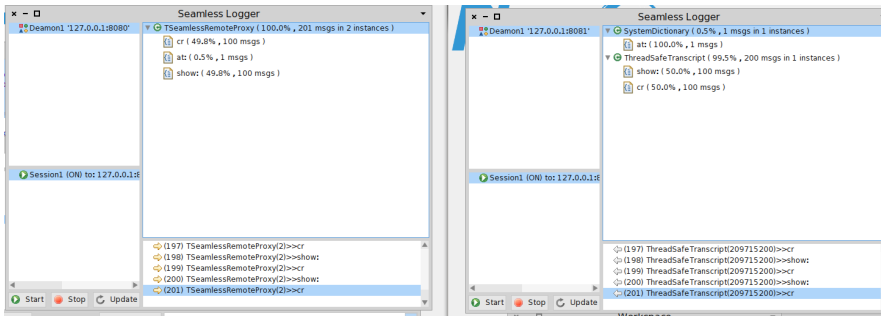

Figure 1.3: Remote Printing of Proxied Objects (LOG).

By sending the message #byValue to a local object (line 4), it will force it to be copied rather than proxied on the remote call it is passed as an argument. Conversely #byValue send to a proxy (ie a remote and not a local object) will return a local copy of it for further processing. Several synonyms exist in Seamless for this facility (ie #asLocalObject, #asRemoteObject etc) but they all do the same thing for both local and remote objects. That is: *Force the serialization of the object in question* in the syntactic context they are used.

Passing Proxies Around

So we 've seen what happens when both terminal instances and composite objects are passed as arguments to remote calls. What about remote proxies themselves, what happens then ? Let's evaluate the following code:

```
1 remoteTranscript := '127.0.0.1:8081' globalAt: #Transcript.
2 (1 to: 100) do: [:i |
3    remoteTranscript show: remoteTranscript; cr.
4 ].
```

This time on line 3 we ask the remote transcript to print itself. The code behaves as expected, we see the transcript on peer2 being printed as a string on itself. This time though behind the scenes (in Figure 1.3) we don't see any implicit proxies created or messages send from both sides. In fact the log when passing proxies as arguments looks identical to the one we saw on Figure 1.1 when passing numbers by value. Why ? Well, it turns out that a proxy passed as an argument to a remote call will just be de-referenced in the other side where it actually resides as a local object.

Return Values

Similarly for return values of remote calls, there are three distinct cases (according to the serialization policies of the peer that responds to the message):

1. A return value is serialized if it is a local terminal instance
2. A return value is proxied if it is a composite object
3. A return value is de-referenced in the other side (since it is a proxy from the calling peer)

Run the following code and watch the values printed in your local transcript to make these distinctions more clear:

```
1 remoteTranscript := '127.0.0.1:8081' globalAt: #Transcript.
2
3 returnValue := remoteTranscript class.
4 Transcript show: returnValue; cr; show: returnValue xxx___isProxy; cr; cr.
5
6 returnValue := returnValue allInstances.
7 Transcript show: returnValue; cr; show: returnValue xxx___isProxy; cr; cr.
8
9  returnValue := returnValue size.
10 Transcript show: returnValue; cr; show: returnValue xxx___isProxy; cr; cr.
11
12 returnValue := (Smalltalk from: '127.0.0.1:8081') globals at: #aRemoteObject put:
       Object new.
13 Transcript show: returnValue; cr; show: returnValue xxx___isProxy ; cr; cr.
```

```
<OUTPUT>

ThreadSafeTranscript
true

{Transcript}
true

1
false

an Object
false
```

On line 1 as previously we retrieve the remote transcript and then on line 3 we send the message #class to it. We print the return value and ask if it is a proxy. The return value is the class *ThreadSafeTranscript* (a composite object from the other side) so it is indeed a proxy (*true*). Then on line 6 we ask from

this remote class all its instances and print them locally. The answer is an ordered collection that includes the remote transcript *{Transcript}* and since this is a composite object it exists on peer1 as a proxy (*true*). Then on line 9 we ask this remote ordered collection for its size. Since size returns a number and numbers are terminal instances the size is returned by value (*false*). Finally on line 12 we create a new entry on the remote System Dictionary storing a local object (from peer1). As we saw earlier this local object will be *implicitly* proxied on peer2. But, when the remote call #at:put: will return the value stored (*an Object*), this value will be a de-referenced local object (*false*) that is also being proxied on the other side.

To sum-up the following Table gives you all the possibilities for both arguments and return values for a two-peer communication:

Type	Distribution
Terminal Instance	**By Value** (Serialized/Copied)
Composite Object	**By Reference** (Proxied)
Proxy	**De-Referenced** (Proxied Object from the other side)

1.3 Between Ping-Pong and the REST

Armed with the experience of our previous example we now move on to a slightly more complicated case in order to understand more about the trade-offs of implicit proxyfication and traffic. Our goal is for the reader to be able to reason with ease about the distribution (either explicit or implicit) of objects and structure his or her code accordingly.

After that we will have a short discussion about the well-documented caveats of *transparent distribution* [6] and how this model compares to - what can be seen as - the the other end of the communication spectrum, namely that of stateless one-shot requests (REST).

Let's Play Ping-Pong

In the first code snippet below (which you can find in the Examples subpackage of Seamless) we see the localPingPong method of the TutorialExample class. This method on line 2 creates a new instance of the PingPong class and sends the message #decrement:using: to it.

```
1 TutorialExample class>>localPingPong
2   PingPong new decrement: 100 using: PingPong new.
```

[6] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7628

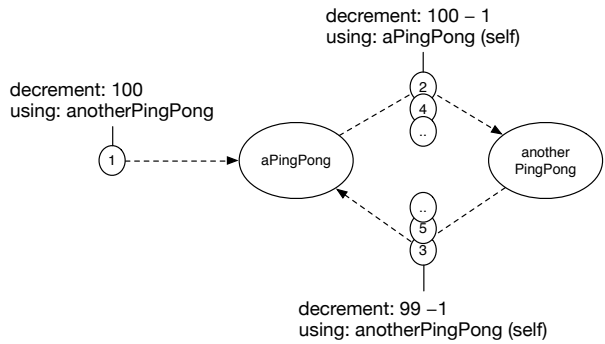

Figure 1.4: Mutally recursive calls between ping-pong instances.

As seen in Figure 1.4 the ping-pong instance through the #decrement:using: method is responsible for decrementing the number *aNumber* which it receives as the first argument through mutual-recursion with another ping-pong instance (whence the name).

Inside the #decrement:using: method (seen below), the number is decreased on line 6 as it is passed back and forth between the two instances, while on line 11 the number of each step is printed. Printing is done after the mutual recursive calls with each executing context (100 contexts in this example) printing the number it received before returning.

```
1 PingPong>> decrement: aNumber using: aPingPongInstance
2
3   aNumber = 0 ifFalse: [
4
5       aPingPongInstance
6           decrement: aNumber − 1
7           using: self.
8
9   ].
10
11  Transcript show: aNumber; cr.
```

If you open a Transcript and execute *TutorialExample localPingPong* you will - as expected - see the count-down from 100 being printed on your screen.

This rather abstract and boring example can become more intresting if one of the two instances is remote. This is achieved on lines 2 and 3 of the #pingPong method in the TutorialExample class (see below) where a local PingPong instance receives a remote instance of the same class as a second

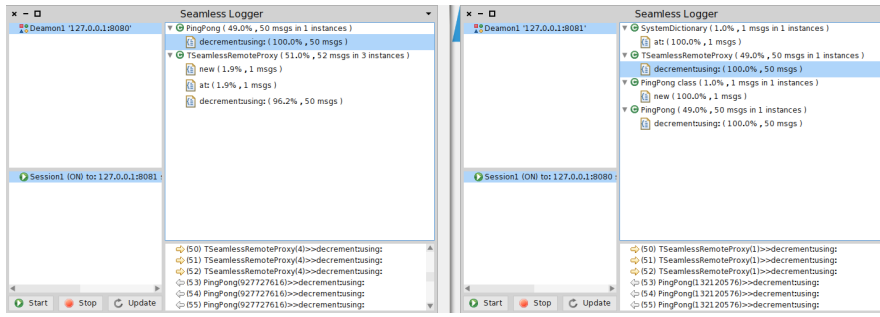

Figure 1.5: The Log of the PingPong Example.

argument.

```
1 TutorialExample class>>pingPong
2   remotePingPongClass := ('127.0.0.1:8081' globalAt: #PingPong).
3   PingPong new decrement: 100 using: remotePingPongClass new.
```

To run this example set-up peer1 and peer2 as previously and open the Transcripts on both sides. You should see that now - in contrast with the local version - only even numbers are printed on peer1 while odd numbers are printed on peer2.

In this case the hundred executing contexts which print the results before returning are coupled between the two machines. Naturally the first call to the local pingPong instance on line 3 below *will wait for all contexts in both machines to return before printing 100*. This is also seen in Figure 1.5 where we log the communication between the two machines.

What we should note here is that one *explicit* remote call (on Line 3 on the code-snippet above) not only generated implicit proxyfications and traffic, but this implicit traffic was heavily coupled between the two machines. In essence the execution time of this one *explicit* remote call, includes:

- it's own execution time and it's own communication overhead
- PLUS the communication overhead that the subsequent **coupled** 100 contexts generated

Now these kind of scenarios can easily appear under *naive* distribution coding. Now although the ping-pong example runs pretty smoothly in Seamless for a lot more than 100 contexts, more coupled and more complicated examples will generate even more implicit traffic. These coupled scenarios can introduce unacceptable slow-downs (latency) to your application or bring application resources (memory, bandwidth) to their limits.

Where is the Ball?

In our particular example even serializing on-demand our objects (ping-pong balls) will not *exactly* help us. That is if we intend to leave the rest of the code unchanged. To illustrate this consider the following two examples below. On the first example we have changed the pingPong method we saw earlier by sending the message #asLocalObject to the remotePingPong instance we have created. This message send will actually return a copy of the remote pingPong instance, effectively serializing the proxied object. This will in turn result in a local-only execution of the #decrement:using: method, since now both ping pong instances reside on the same machine. You can confirm this by running the following code and watching the whole sequence of numbers being printed only on peer1.

```
1 TutorialExample class>>pingPong
2   remotePingPongClass := ('127.0.0.1:8081' globalAt: #PingPong).
3   PingPong new decrement: 100 using: remotePingPongClass new asLocalObject.
```

Alternatively you could force the otherwise implicitely generated proxy from peer1 to be passed by value as follows:

```
1 PingPong>> decrement: aNumber using: aPingPongInstance
2
3   aNumber = 0 ifFalse: [
4
5       aPingPongInstance
6           decrement: aNumber − 1
7           using: self byValue.
8
9   ].
10
11  Transcript show: aNumber; cr.
```

The only thing that has changed here is the *byValue* directive on line 7 which will force the local ping pong instance of peer1 to be sent by copy to peer2. You can confirm this by running the code and watching the whole sequence of numbers (except the first) being printed only on peer2.

In both examples above by serializing the distributed objects - despite de-coupling the communication - we actually changed the *semantics of the application* since now we don't get the same side-effects as in the first scenario (even numbers on one peer and odd numbers on the other).

Of course the solution [7] to this *coupling* problem (if we hypothesize that our intention is to print odd and even numbers on two different peers) is to change the code itself to look more like the first *Hallo Transcript* example on

[7] Apart from using optimizations like futures or batch communication (see concluding Section)

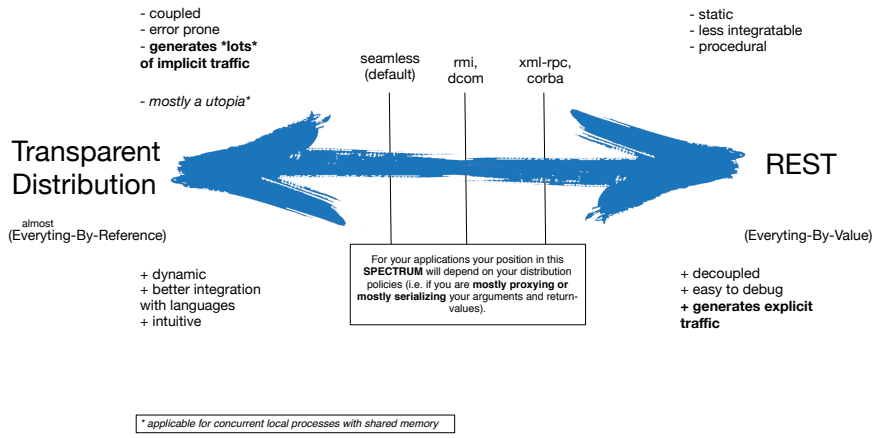

Figure 1.6: A (over-simplified) view of the spectrum of communication middleware solutions in terms of distribution policies and implicit traffic.

Section 1.2. That example just prints numbers in the other side in one-shot remote message-sends by passing arguments by copy.

The (Simplified) Big Picture

We generalize and illustrate this situation in Figure 1.6. It turns out that the more objects or classes of objects you pass by value (like the first *Hallo Transcript* example) the more *closer* you are to a communication paradigm like REST where everything is passed as a copy and each call is a *one-shot* request against a simple communication API (that is uncoupled and does not generate implicit traffic). On the contrary the more you rely on the implicit proxyfication of objects between calls (like in the ping-pong example) the more closer you are to *transparent* distribution and its documented caveats (strong coupling, latency, hard to debug). On the other hand REST is more static and feels like a procedural communication API rather than an object-oriented one while transparent distribution is fully object-oriented and intuitive. But when this transparency is taken to extremes it can be seen as *mostly utopic*. [8]

The idea behind Seamless is that you should be able to play across the whole spectrum while you are prototyping and fine-tune your own code by profiling and considering the trade-offs. Even more so you may want to use Seamless to craft useful abstractions that enforce certain paradigms to your

[8] With the exception of transparent distribution over shared local memory (see concluding Section)

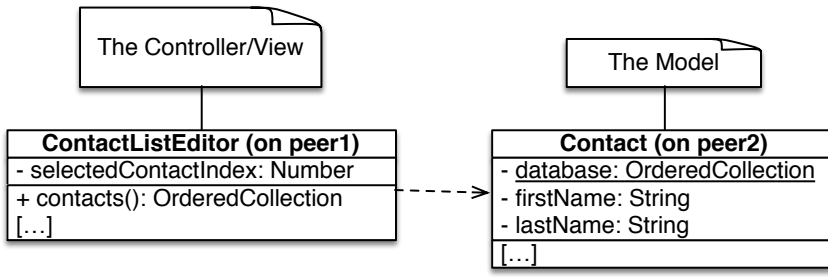

Figure 1.7: Main classes of the Contact manager application of Pharocasts.

code.

1.4 Practice Makes Perfect

In this section we will go over some applications from the Seamless-Examples package, that you should play around with to get more comfortable with the framework.

Ok, ok: Contact the Beatles

First on our list is the ContactListEditor app from PharoCasts [9]. We will be repurposing the app using Seamless to make it browse and edit contacts from a remote peer.

The ContactListEditor is a local application for contact management that is aimed at showcasing Polymorph. The two main classes of the app are shown in Figure 1.7. It has a simple model-view architecture. We will be running through Seamless the view (seen on the left of the figure) on peer1 and the model (on the right of the figure) on peer2. The view consists of the ContactListEditor class, which remembers the currently selected contact (*selectedContactIndex* iv) and retrieves all available contacts through the #contacts method. These contacts are actually stored on the class-side variable of the class Contact (*database* iv), with each Contact instance holding a firstName and a lastName string.

The class ContactsDatabaseExample on the Seamless-Examples package provides some helper methods to load example data on each peer for the

[9] http://www.pharocasts.com/2011/02/pharo-gui-with-polymorph.html

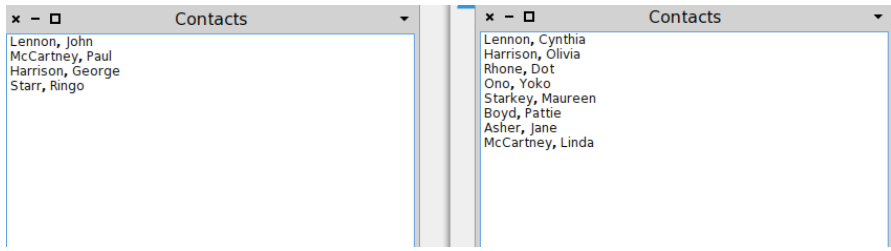

Figure 1.8: The Contact Manager with the Beatles (peer1) and the Beatles Wifes (peer2).

application. Go ahead and evaluate the following code on peer1 and peer2 respectively.

```
ContactsDatabaseExample loadBeattles (on peer1)
...
ContactsDatabaseExample loadBeattlesWifes (on peer2)
```

Then by evaluating:

```
ContactListEditor open. "the local editor"
```

on each peer you should be seeing something similar to Figure 1.8. This is the local behavior of the app (as it was originally intented) with our sample data (ie the Beatles on peer1 and Yoko Ono and her friends on peer2).

Now let's evaluate the following code on peer1, to browse the contacts of peer2 remotely:

```
Contact database: (Contact from: '127.0.0.1:8081') database.

ContactListEditor open. "browsing contacts from remote peer"
```

By now you should be comfortable by what has been done here. On the first line of the code-snippet we pointed the Contacts database to peer2 and then on the second line we re-opened the editor. Now Yoko (and the rest of the Beatles wifes) is shown on peer1 although they are stored on peer2. [10]

Figure 1.9 shows the results of this remote re-purposing and the SeamlessLogger. Go ahead and browse or edit the contacts and also try to move around the window. If it feels a bit sluggish, update the SeamlessLogger to see why. The ui (morphic in this case) in the view is sending messages to

[10] Of course we could have achieved that in a more elegant way, by extending the view to be able to browse more than one databases. In this case we wanted to leave the original application code as unchanged as possible.

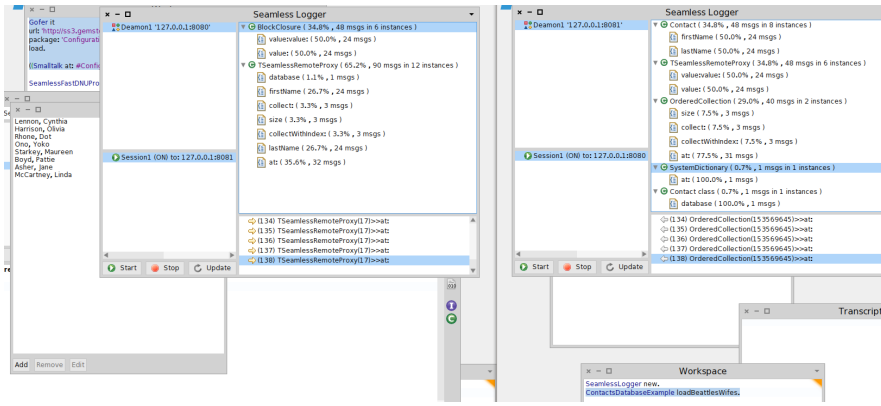

Figure 1.9: Loading the Contacts of a Remote Peer.

your model (stored on the other side) even while you are moving the window !

Of course there are many ways to change that, with the first one being to *control when morphic asks the model for updates*. For example you can adopt a Model-Model-View-Controller pattern in your code (for more on this read here [11]). What this means essentially is that *what you see* in the ui will be the *Application Model* or the *UI Model*, which will be different from the *Domain Model* or *Back-End Model* (the Contacts database in this case). If you have this distinction in your code, morphic will be sending messages to the first model which is by definition local, while the second model can be distributed (like in our case) without Morphic getting in the way.

Another way would be to use a more *rest-like* approach by using Seamless as we described earlier. This is seen in the following code-snippet. Go ahead and try it:

```
Contact database: (Contact from: '127.0.0.1:8081') database asLocalObject

ContactListEditor open. "browsing contacts from remote peer"
```

As you may have guessed what we did here is to *on-demand serialize* our request to the other side by bringing all remote contacts locally. Of course we brought the whole of the contacts database in this example (since it is small). In a real-world scenario you would be probably doing this incrementally (ie bringing the first n-contacts and then using paging to navigate). Serializing means that editing, browsing etc does not generate additional requests but also that now it is your responsibility to sync the local copy to the other side.

[11] http://c2.com/cgi/wiki?ModelModelViewController

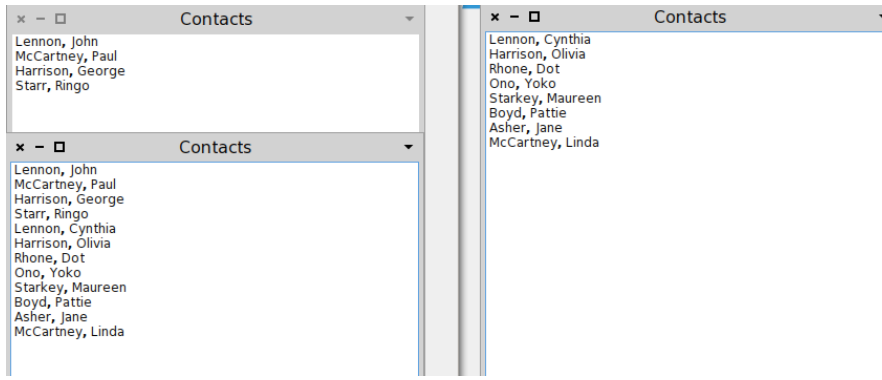

Figure 1.10: Mixing Local and Remote Contacts (Crude Data-Balancing).

Finally on Figure 1.10 we see something more exotic that you can do with the contact manager example. This is a *crude* data-balancing case. Go ahead and try on peer1 the following code-snipet:

```
ContactsDatabaseExample loadBeattles.

Contact database: (OrderedCollection new
        addAll: Contact database;
        addAll: (Contact from: '127.0.0.1:8081') database;
        yourself).

ContactListEditor open.
```

Here we first restored the Beatles on peer1 and then added both the local Beatles and their remote wifes (as proxies) on the local db. The result as seen in Figure 1.10 is a database that *seamlessly* (pun intended) integrates contacts from two different machines.

Ok, ok: Make me a Remote Tester

Next stop is remote testing. Take a look at the class TestRunnerExample and its #testCategoryNamed:at: method below. From lines 1 to 5 we run the tests of a specific category as you might have expected. That is we select first the category (line 3) then all its classes (line 4) and finally we run the tests (line 5).

```
TestRunnerExample»testCategoryNamed: aCategoryName at: aRemoteAddress

   | tr |
```

```
1  tr := (TestRunner from: aRemoteAddress) new.
2  tr
3      categoryAt: (tr categoryList indexOf: aCategoryName) put: true;
4      selectAllClasses;
5      runAll.
6
7  "difference on inspection between local and remote objects"
8  tr result inspect.
9  tr result asLocalObject inspect.
10 Smalltalk tools inspector inspect: tr result.
```

The difference of course with a local such scenario is on line 1 where instead of the local TestRunner we fetch a remote one (whose address is described by *aRemoteAddress* argument).

Now the more crucial part of this example is on lines 7 to 10 where we inspect the test results in three different ways. If you have your two peers set-up evaluate the following code to run the *ProfStef* tests of peer2 from peer1 and check the results:

```
TestRunnerExample new
    testCategoryNamed: #'ProfStef-Tests'
    at: '127.0.0.1:8081'
```

As seen in Figure 1.11 there are three inspectors opened, one in peer2 and the other two on peer 1. Line 8 is responsible for the inspector on peer2 (on the right). This inspector opened on the target image (that is peer2 where the tests where run) because the #inspect message was send to a proxy ! This is the proxy object returned by the message #result sent to another proxy (namely *tr*) which we retrieved on line 1.

What if now we wanted to see the results remotely on peer1 ? This is done on line 10 where we open the local inspector of peer1 "on" the proxy instead of asking the proxy itself to be inspected (as we did on line 8). Most of the traffic generated (that is seen on the logs of Figure 1.11) originates by this inspector, which is contacting the other side.

Finally, the best way - in this case - to browse the remote results on peer1 is seen on line 9. Since test results are actually computed only once and do not change (ie do not have to be in sync with the other side) we serialize them (bringing them to peer1) where now the message #inspect is send to a local object.

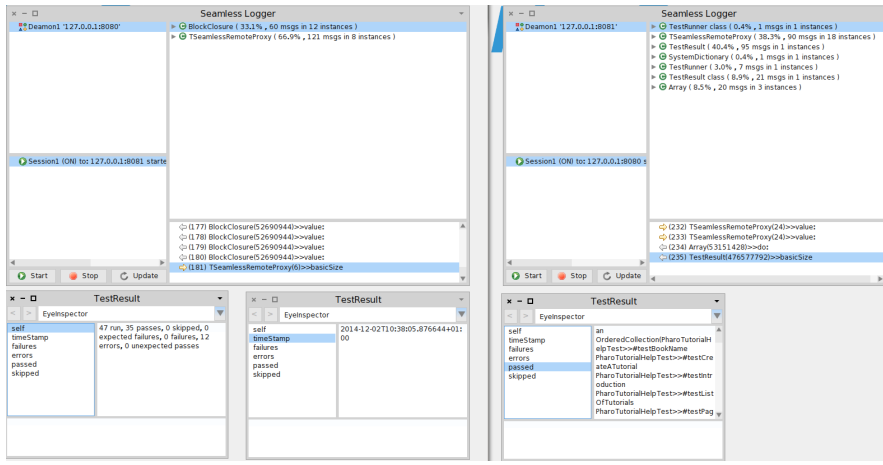

Figure 1.11: Different ways to to distribute the test results.

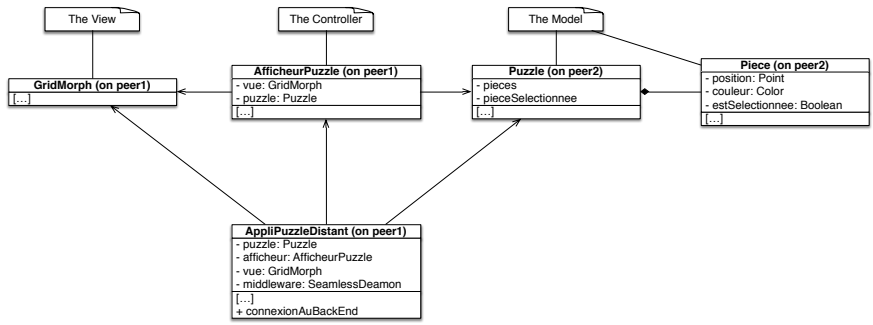

Figure 1.12: Main classes of the "puzzle-game" application of Car-Mines.

Ok, ok: Make me a Distributed Game

Last (but not least) on our list is a puzzle-game from the Car-Mines lab of Ecole-des-Mines de Douai [12] . We will be using Seamless to connect to the game's *back-end* from a remote peer who runs the puzzle *front-end*. The main classes of the game are seen in Figure 1.12.

The game has an MVC architecture with the class GridMorph (on the left side) on the role of the view, the class AfficheurPuzzle on the middle on the role of the controller and the classes Puzzle and Piece as the model. Instances of Puzzle hold a list of pieces and the current selected piece among them,

[12] http://car.mines-douai.fr/

with each piece having a position (Point), a couleur (Color) and a Boolean value describing if the piece is selected or not. The controller has references to both the view and the model. Finally the class AppliPuzzleDistant controls the middleware (a SeamlessDeamon) and assembles the view, the controller and the model together.

You can run and play around with the game by evaluating the following code on peer2 and peer1 respectively:

```
BackEndAppliPuzzleDistribue demarrer. "peer2"
[...]
AppliPuzzleDistant demarrer. "peer1"
```

Figure 1.13 shows the puzzle game in action with the SeamlessLogger profiling it. The following method (#connectionAuBackEnd) which is called from within the #demarrer method we called earlier is where all the magic happens:

```
AppliPuzzleDistant>>connexionAuBackEnd
   | classePuzzleDistante |
1  classePuzzleDistante := (middleware connectOn: '127.0.0.1:1111') at: #Puzzle.
2  puzzle := classePuzzleDistante new.
3  afficheur := AfficheurPuzzle new.
4  afficheur puzzle: puzzle vue: vue.
5  vue onClickDo: [:x :y|
6      Transcript show: 'sending'; cr.
7      puzzle inverserSelectionPieceX: x y: y.
8  ]
```

On line 1 the remote model is retrieved via the middleware (a SeamlessDeamon) and a new puzzle is created (line 2) which is proxied. On line 3 a new controller is instantiated and is initialized with the view (this is a GridMorph as the one shown on Figure 1.13) and with the remote puzzle. Then finally on lines 5 through 8 the view is instructed through the #onClickDo: method to invert (on a left-click) the currently selected piece with the piece on the click's (x,y) co-ordinates.

What is interesting to note about the puzzle game (as shown in the log of Figure 1.13) is that apart from the *core-logic* remote calls of the game (the #inverseSelectionPieceX:y message we have seen) there is a significant amount of traffic generated just by connecting the local controller with the remote model on line 4. This is to be expected since in this example too the *application* and the *domain* model are the same. Nevertheless the game runs very smoothly under Seamless since the return values of most of the messages send to the backend (#x, #y, #couleur etc) - which we see in Figure 1.13 - are serialized automatically by the framework (being terminal instances such as Points and Colors).

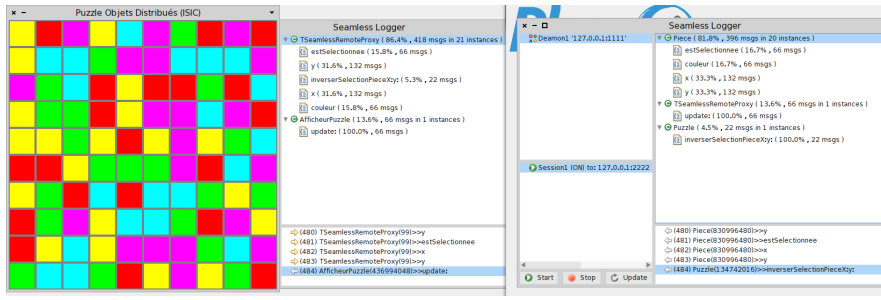

Figure 1.13: Front-End and Log of the Puzzle Game.

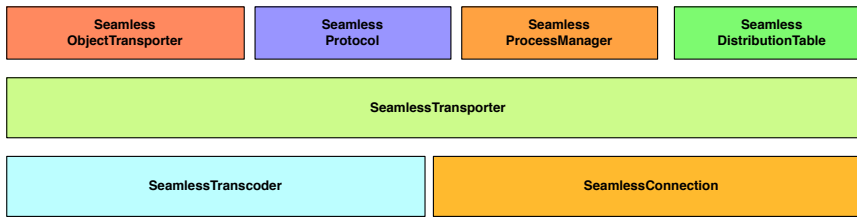

Figure 1.14: Low-level architectural overview of Seamless.

1.5 How It All Works?

Lower-Lever Overview

On Figure 1.14, we provide an overview of the low-level communication infrastructure of Seamless:

SeamlessConnection Low-level bidirectional asynchronous event-driven communication abstraction.

A seamless connection has the same api regardless of the underlying communication protocol or medium. Currently Seamless can operate both over plain sockets with its own protocol and (slower) over http.

SeamlessTranscoder This is our marshaller which is responsible for serializing and materializing information, passed through the connection.

Different marshalers can support different transcoding algorithms. Currently Seamless operates by wrapping a fast binary marshaller (Fuel), but

also a simpler yet more verbose string marshaller has been tested. Other options could include serializing to xml, json etc

SeamlessTransporter The concrete subclasses of this abstract class handles the actual communication between peers.

Each concrete transporter knows about the communication channels supported by the peers (protocols, types of sockets etc) and establishes an appropriate SeamlessConnection between them.

SeamlessObjectTransporter This is an OO abstraction for the transporter.

Instead of sending or receiving bytes, plain-text, xml etc higher-level components of the Seamless framework exchange objects through this class. These objects are instances of one of the SeamlessProtocol classes.

SeamlessProtocol This is a whole hierarchy of classes, that defines an open object protocol.

As we saw, connected peers with Seamless exchange objects (through the object transporter). These objects are instances of one of the SeamlessProtocol classes that contain both data (other objects) and meta-data (describing the semantics of the object exchange, message-passing infromation etc).

SeamlessProcessManager While a SeamlessConnection is asynchronous by itself the ProcessManager can create its own blocking strategy.

The SeamlessProcessManager is listening to the asynchronous communications and suspends or resumes requesting processes on-demand.

SeamlessDistributionTable This is an actual reference table keeping track of remote references, as well as local references of objects from other peers.

Remote referencing in Seamless is also adaptable with two tested available implementations. The first one uses Ghost which is a uniform, lightweight and stratified general purpose proxy model, while the second one is more specialized and is based on shadow classes.

Higher-Lever Overview

Figure 1.15 provides an overview of the high-level communication components of Seamless. These depend on the low-level communication infrastructure, through *SeamlessSession* which has a one-to-one relationship with the *SeamlessConnection* on Figure 1.14.

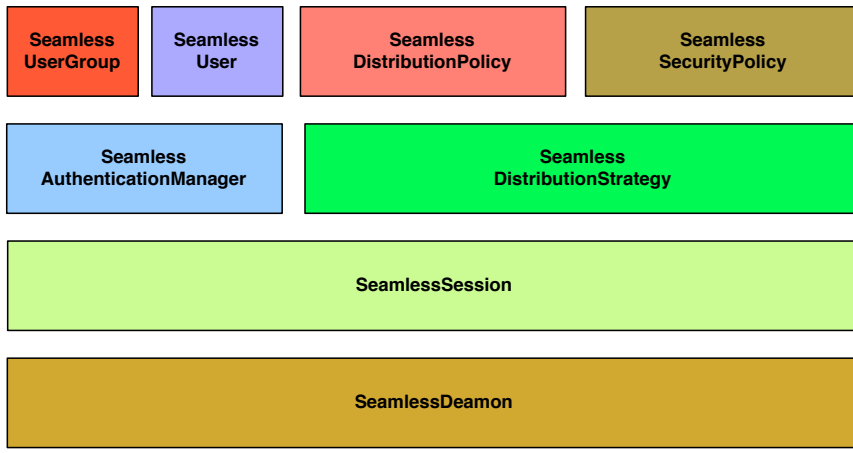

Figure 1.15: High-level architectural overview of Seamless.

SeamlessDeamon This class defines methods for the orchestration (assembling) and initialization of our middleware, acting more or less as a Factory.

Deamons can handle multiple connections (see also SeamlessSessions) to different peers and each environment can have multiple deamons (with different adaptations) running in parallel.

SeamlessSession This is a high-level view of a SeamlessConnection that is established upon successful authentication on both sides.

SeamlessAuthenticationManager A simple authentication manager (see users and groups below).

SeamlessUserGroup A user group is a named set of users that is associated with a specific distribution strategy.

SeamlessUser This is a standard/user password pair.

SeamlessDistributionStrategy A distribution strategy combines a distribution policy with a security policy (see below) to make decisions about whether or how communicaton will proceed.

SeamlessDistributionPolicy This class (through its concrete subclasses) decides how specific objects or group of objects will be distributed among peers.

Options can include: full serialization, shallow serialization, proxying, etc..

SeamlessSecurityPolicy The concrete subclasses of this abstract class are responsible for authentication and for restricting access (either message sending or distribution) for specific instances or whole classes of objects.

1.6 Seamless Papercuts

It is the little things in life that will make you mad. Here are some do's and dont's while using Seamless. These *papercuts* will incrementally go away as we advance to more stable releases.

NEW: *Latest Development Branch Solves UIProcess, Workspace and Morphic unresponsiveness under heavy messaging*

The UI Process

All ui applications of Pharo are naturally *event-driven*. What this means is that their application logic is *triggered* by input events such as keyboard-strokes and mouse-clicks. Upon the invocation of such events application callbacks are evaluated from within the UI Process. Essentially this results in all front-end applications being handled by a single green thread, unless otherwise instructed by the programmer of the callback.

This situation can cause the ui to be unresponsive, if for example a single morph is taking too long to compute its callback (try interactively evaluating *(Delay forSeconds: 10) wait* from any text morph to see what happens). This *papercut* although not specific to Seamless can create frustration when you are proxying application models and feeding them directly to morphs. Using Seamless with futures will help in this case (see below) but the more general solution that we intent to integrate is to equip the UI-Process with sub-threads (either globally or per application) as to allow the ui to respond to events while a sub-ui-thread still computes the *slow* callback of a morph.

The Workspace

A somewhat related papercut involves the workspace. You will be probably experimenting with Seamless by evaluating stuff through the workspace or elsewhere so you should remember that [ ] forking your *slow* do-its might help avoid unresponsiveness by the ui. Morover the workspace is currently being particularly *mean* to proxies of all kinds (distributed or otherwise)

by sending them all shorts of messages for syntax-highlighting and internal book-keeping. By fixing the UI-Process and using memoization for this workspace noise (or even making the workspace aware of proxies) this papercut will be resolved.

Morphic

This is again not specific to Seamless, but can nevertheless hurt your distributed adventures. Morphic sends a lot of messages to your models to update, some of them can be automatically memoized by Seamless (we are working on that) others not. As previously stated using an MMVC pattern and controlling the updates yourself between the *application* and the *domain* model resolves this issue. Ultimately though we will be looking more closely at Morphic (as well as Spec and Glamour) to see if they can provide some kind of abstraction (such as an explicit update policy) to help you do that more easily.

Image-Save

Please save your code on a repository or on the local-cache and reload your projects on start-up on a clean image. This is a good practice in general, trust us. In the future we may review the ability to resume previously *image-saved* connections. Right now (to keep us all sane) we perform aggressive clean-ups (on deamons, sessions and sockets) upon every start-up and shut-down.

Exceptions

Once we move to a stable release we will be porting remote reflection facilities (from the Mercury prototype) and full-blown exception handling for your remote exceptions. Right now Seamless only forwards back to the original caller (that may be many remote calls coupled) a SeamlessRemoteException that is less informative than you might expect. Stay tuned.

1.7 Where Do We Go From Here?

Short-Term

- More Documentation / Testing / Refactoring: The Seamless code-base requires some additional love. Being for most of its life time an engineering part of a larger research prototype *there can be dragons here*.

- Papercuts: See Section 1.6 above. A stable release should not suffer from these annoyances.

- Shared-Memory: As seen in Figure 1.6 there is a special and very interesting case where the *mostly utopic* transparent distribution end of the communication spectrum makes sense. We are currently working on that. Current experiments in development branch include NanoMsg and pure posix shared-memory transporters.

Mid-Term

- Futures / Batch-communication / Memoization: There is a large list of optimizations for systems like Seamless from both academia and industry. We will take it one step at a time. After this is done comparing to systems like RMI and DCOM both in terms of features and performance should be the next logical step.

- Low-level Profiling: The underlying low-level communication solution of Seamless is quite descent for a stable release. We need numbers and low-level profiling (on sockets) to see what can go even better here.

- Closer integration with Fuel: Lots of great things can happen on this front, by being able to migrate processes, classes or incrementally-serializing remote object graphs.

- More Monitoring: The SeamlessLogger has proven to be a very convenient tool. We need more of those tools for interacting with the SeamlessProcessManager, the SeamlessDistributionTable, viewing RemoteExceptions etc.

- Distributed-GC: We have the hooks for implementing simple distributed-gc algorithms and a long list of related bibliography. Quite doable at this stage but also a whole field on its own.

- Remote Reflection: Bringing parts of our research prototype for remote reflection to enough maturity for community-use is one of Seamless *raison d' etre*

- Field Applications: Field applications where Seamless is used daily and from which we can get constant feedback.

1.8 Summary

This Chapter presents Seamless a reflective communication middleware for Pharo that aims to facilitate the prototyping of distributed applications. After a short introduction on how to install and get started, we present all the details of remote messaging under Seamless. Specifically how to fetch remote references, pass arguments by value or by reference and retrieve return values. Moreover we show how to set-up distribution policies and change those same policies on the fly to fit your needs. We introduce you to the SeamlessLogger which is used for logging and profiling of remote messaging and discuss the trade-offs between explicit and implicit proxying and traffic. Subsequently we walk through three application examples from the SeamlessExamples package and give you an overview of the internal workings of the framework. Finally we detail a list of do's and dont's to avoid common caveats associated with Seamless and discuss some short-term and mid-term future perspectives.